# Efficient Constraint-Based Dynamic Strategies For Generating Counterexamples*

Nguyen Le Vinh
Hélène Collavizza
Michel Rueher
University of Nice – Sophia Antipolis / CNRS
2000, route des Lucioles - Les Algorithmes
BP 121 - 06903 Sophia Antipolis Cedex - France
lvnguyen@polytech.unice.fr
helen@polytech.unice.fr
michel.rueher@gmail.com

Samuel Devulder
Thierry Gueguen
Geensys
120 Rue René Descartes
29280 Plouzané - France
samuel.devulder@geensys.com
thierry.gueguen@geensys.com

## ABSTRACT

Checking safety properties is mandatory in the validation process of critical software. When formal verification tools fail to prove some properties, testing is necessary. Generation of counterexamples violating some properties is therefore an important issue, especially for tricky programs the test cases of which are very difficult to compute. We propose in this paper different constraint based dynamic strategies for generating structural test cases that violate a postcondition of C or JAVA programs. These strategies have been evaluated on standard benchmarks and on real applications. Experiments on a real industrial Flasher Manager controller and on the public available implementation of the Traffic Collision Avoidance System (TCAS) show that our system outperforms state of the art model checking tools and constraint based test generation systems.

## Keywords

constraint-based test data generation, counterexamples, search strategy

## 1. INTRODUCTION

In modern critical systems, software is often the weakest link. Thus, more and more attention is devoted to the software verification process[2]. In practice, software verification includes formal proofs, functional and structural testing, manual code review and analysis. When formal verification tools fail to prove the postconditions of a program, this job is often done by hand in practice. Moreover, formal verification can only prove property violation or satisfaction on an abstract model which does not capture all implementation features, and which is often of limited size. Thus, testing is always necessary, and automatic generation of counterexamples violating a property is an important issue. The point is that such test cases may be very difficult to generate on tricky programs and real applications. In real time applications, generating test cases for realistic time periods is still an open challenge.

In this paper we propose new constraint based dynamic strategies for generating structural test cases that violate a postcondition of C or JAVA programs. Our approach is based on the following observations:

- when the program is in an SSA-like form[1], a faulty path can be built in a dynamic way. In other words, we do not need to explore the CFG (Control Flow Graph) in a top down (or bottom up) way but we can just collect compatible blocks in a non-deterministic way.
- A significant part of the program may have no impact on a given property[2]. Especially, parts of program that neither contain variables of the required property, nor variables connected to variables of the required property, can be ignored when we search for a counterexample.

The Dynamic Postcondition-Variables driven Strategies ($DPVS$) we have defined take advantage of these two observations. Informally speaking, $DPVS$ works with a constraint store $S$ and a queue of variables $Q$. $Q$ is initialized with the variables in the property of the postcondition for which we are searching a counterexample, whereas $S$ is initialized with the negation of this property. As long as $Q$ is not empty, $DPVS$ removes the first variable $v$ and searches for a program block where variable $v$ is defined. All new variables (except input variables) of this definition are pushed

*This work was partially supported by the ANR-07-SESUR-003 project CAVERN and the ANR-07 TLOG 022 project TESTEC.

[1]SSA (Static Single Assignment) form is an intermediate representation used in compiler design: it is a semantics-preserving transformation of a program in which every variable is assigned exactly once [8].

[2]We assume here that the postcondition is a conjunction of properties.

.

on Q. The definition of variable $v$ as well as all conditions required to reach the definition of $v$ are added to the constraint store $S$. If $S$ is inconsistent, *DPVS* backtracks and searches for another definition; otherwise the dual condition to one added to $S$ is cut off to prevent *DPVS* from loosing time in exploring trivially inconsistent paths. When $Q$ is empty, the constraint solver searches for an instantiation of the input variables that violates the property, that's to say a counterexample.

We have explored different dynamic ordering of the queue of variables: FIFO, LIFO, heap ordered by the level of the variable in the compacted control flow graph. We obtained the best results with the LIFO ordering and the heap. Heap ordering promotes variables that are defined early in the program. The intuition behind the LIFO strategy is the following one: collecting as much information as possible on a given variable enforces the constraints on its domain and reduces the search space. In other words, we have a great chance to detect inconsistencies as early as possible by following the thread of a variable which occurs in the postcondition.

We also tried different combinations of finite domain constraint (CP) solvers and linear programming (LP) solvers. None of these combinations is consistently better than others but the results are somewhat predictable: when the domains of the variables are small, CP behaves better; when numerous linear expressions occur, LP is usually better.

We evaluated *DPVS* on standard benchmarks as well as on two real applications:

- The well-known public available implementation of the Traffic Collision Avoidance System (TCAS);
- A real industrial application, called Flasher Manager, a controller that drives several functions related to the flashing lights of a car.

On these real applications, *DPVS* outperforms state of the art model checking tools and constraint based test generation systems.

## 1.1 Related work

Modern path-oriented structural test data generators (eg., PathCrawler [21], Dart [12] and CUTE[3]are based on path selection, symbolic execution and concolic execution. They are very efficient for test coverage monitoring and measurements but they are not designed to find the test data that violate some property; of course, they could do it by performing an exhaustive search but this would be very costly.

Bounded model-checkers [10] like Blast[4] or CBMC[5] can find counterexamples to properties over C programs [11, 16]. Bounded model checkers transform the program and the postcondition in a big formula and use SAT and SMT solvers to prove that the property holds or to find a counterexample.

Constraint Logic Programming (CLP) was used for test generation of programs (e.g., [15, 17, 22, 1]) and provides a nice implementation tool extending symbolic execution techniques [3]. Gotlieb et al showed how to represent imperative programs as constraint logic programs: InKa [15] was a pioneer in the use of CLP for generating test data for C programs. Denmat et al developped TAUPO, a successor of InKa which uses dynamic linear relaxations [9]. It increases the solving capabilities of the solver in the presence of non-linear constraints but the authors only published experimental results for a few academic C programs.

Euclide [13] is also a successor of InKa. It has three main functions: structural test data generation, counterexample generation and partial program proving for critical C programs. Euclide builds the constraints in an incremental way and combines standard constraint programming techniques and specific techniques to handle floating point numbers and linear constraints.

CPBPV [5, 6, 7] is a constraint-based framework the goal of which is to verify the conformity of a program with its specification, that is, to demonstrate that the specification is a consequence of the program under the boundness restrictions. The key idea in CPBPV is to use constraint stores to represent both the specification and the program, and to non-deterministically explore execution paths of bounded length over these constraint stores. CPBPV provides a counterexample when the program is not conforming to the specification.

The point is that the search strategies of Euclide and CPBPV are not well adapted for searching a counterexample. Indeed, CPBPV is based on a top down exploration of the bounded feasible paths because it has been designed for partial program verification. Euclide –which has been designed for test data generation– explores dynamically the feasible alternatives but also in a top down way. When the goal is only to find a counterexample on a large and complex program, both strategies may become very expensive. At the opposite, *DPVS* is a dynamic bottom up strategy which has been designed to find counterexamples on tricky programs.

## 1.2 Outline of the paper

Section 2 shows how our approach works on a small example and introduces the new search algorithms we have defined. Section 3 describes the benchmarks and applications we used to validate our approach. Section 4 reports experiments results and presents further research directions.

# 2. DPVS, THE NEW CONSTRAINT BASED SEARCH STRATEGIES

In this section, we first describe in very general terms the principles of our approach and describe the search process on a small example. Then, we detail the search algorithm.

## 2.1 A small Example

Consider a program $P$ with a precondition *pred* and a property *prop* of the postcondition of $P$. Let $G$ be the CFG of $P$. Let $V(e)$ denotes the temporary variables of expression $e$, that's to say all variables which occur in $e$ except the input variables. $def[x, u]$ denotes the definition of variable $x$ in block $u$. As said before, *DPVS* works with a constraint store $S$ and a queue of variables $Q$. For seek of simplicity we consider here only the LIFO ordering of the queue. $Q$ is initialized with $V(prop)$, the temporary variables of property *prop* whereas $S$ is initialized with the constraints of *pred* and the negation of *prop*. As long as $Q$ is not empty, *DPVS* removes the first variable $v$ and search for a block $u$ where variable $v$ is defined and pushes variables $V(def[v, u])$

[3]see `http://osl.cs.uiuc.edu/~ksen/cute/`
[4]see `http://www.cs.ucla.edu/~rupak/blast/`
[5]see `http://www.cprover.org/cbmc`

on Q. The constraints derived from $def[v,u]$ as well as the constraints derived from the conditions required to reach the definition of $v$ are added to the constraint store $S$. If the solver detects an inconsistency in $S$, *DPVS* backtracks and searches for another definition; otherwise the dual condition to one added to $S$ is cut off to prevent *DPVS* from loosing time in exploring trivially inconsistent paths. When $Q$ is empty, the constraint solver searches for a solution, that's to say an instantiation of the input variables that violates property *prop*. If no solution exists, we have to backtrack.

Now, let us illustrate this process on a very small example, the program *foo* displayed on Figure 1. This program has two postconditions: $p_1 : c >= d + e$ and $p_2 : f > -b * e$.

The CFG of the program *foo* is displayed in Figure 2. Note that the program has been transformed in an SSA-like form[6]. Before searching a counterexample, we compute a compact form of the CFG where the nodes that are trivially not related to the property to prove are removed (see Figure 3).

Now, assume we want to prove property $p_1$. Figures 4 and 5 depict the paths explored by *DPVS*. The search process first selects node (4) where variable $a_0$ is defined. To reach node (4), the condition in node (0) must be true. Thus, this condition is added to the constraint store $S$ and the other alternative is cut off. At this stage, $S$ contains the following constraints: $\{c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = a_0 \wedge a_0 \geq 0\}$ which can be simplified to $\{a_0 < 0 \wedge a_0 \geq 0\}$. This constraint store is inconsistent and thus *DPVS* selects node (8) where variable $c_0$ is also defined. To reach node (8), the condition in node (0) must be false. Thus, the negation of this condition is added to the constraint store $S$ and the other alternative is cut off. Now, constraint store $S$ contains the following constraints: $\{c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = b_0 \wedge a_0 < 0 \wedge d_0 = 1 \wedge e_0 = -a_0\}$ which can be simplified to $\{a_0 < 0 \wedge b_0 < 0\}$. This constraint store is consistent and the solver will compute a solution, e.g., $\{a_0 = -1, b_0 = -1\}$. These values of the input variables are a test case which demonstrates that program *foo* violates property $p_1$.

This small example illustrates how *DPVS* works. It can also help to understand the intuition behind this new strategy: *DPVS* collects the maximum of information on the variables which occur in postcondition to detect inconsistencies as early as possible; this is especially efficient when a small sub-set of the constraint system is inconsistent.

## 2.2 Algorithm

We now detail algorithm *DPVS* (see algorithm 2.2). As an initial step, we compute :

- $du[x]$: the set of blocks where variable $x$ is defined;
- $anc_c[u]$: the set of ancestors of $u$ which are conditional nodes;
- $dr[u,v]$: a boolean which is true (resp. false) when the condition of ancestor $v$ of node $u$ has to be true (resp. false) to reach $u$.

*DPVS* uses also the following data structures:

[6]Let us recall that we only consider here bounded programs. By bounded program, we mean program where the size of the arrays, and the number of iterations of the loops are bounded. We take advantage of the boundness property to simplify the $\phi-$functions when building the SSA form.

```
void foo(int a, int b)
1.  int c, d, e, f;
2.  if(a >= 0) {
4.      if(a < 10) {
5.          f = b − 1;
6.      }
7.      else {
8.          f = b − a;
9.      }
10.     c = a;
11.     if(b >= 0) {
12.         d = a; e = b;
13.     }
14.     else {
15.         d = a; e = −b;
16.     }
17. }
18. else {
19.     c = b; d = 1; e = −a;
20.     if(a > b) {
21.         f = b + e + a;
22.     }
23.     else {
24.         f = e ∗ a − b;
25.     }
26. }
27. c = c + d + e;
28. assert(c >= d + e); // property p1
29. assert(f >= −b ∗ e); // property p2
```

**Figure 1: Program foo**

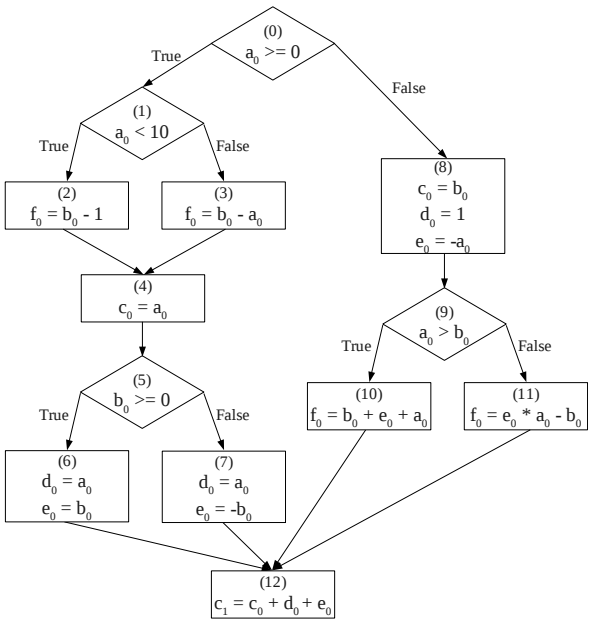


**Figure 2: CFG of program foo in SSA-like form**

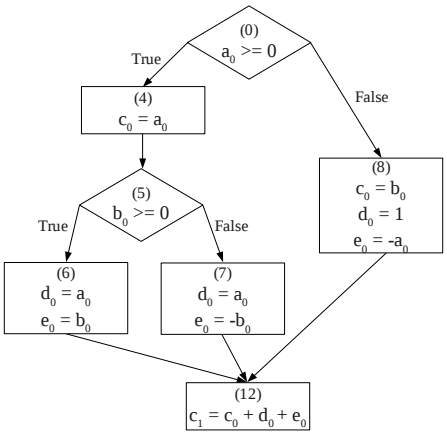


**Figure 3: Compacted CFG for $p_1$**

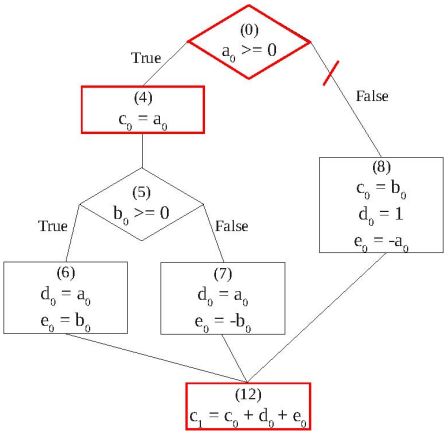


**Figure 4: Search process for $p_1$, step 1**

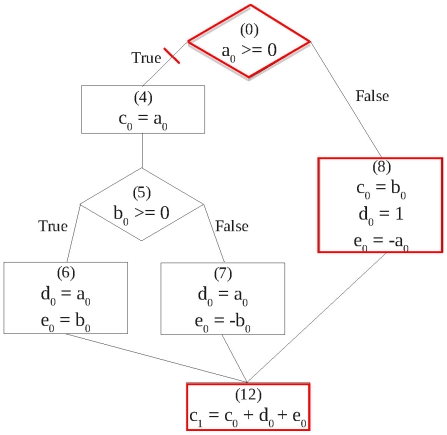


**Figure 5: Search process for $p_1$, step 2**

- $M$: set of marked variables (a variable is marked if it has already been put into the queue); $M$ is initialized with $\emptyset$ ;
- $S$: the constraint store which is initialized with $const(pred \wedge \neg(prop))$ where $const$ is a function that transforms an expression in SSA form into a set of constraints;
- $Q$: the set of temporary variables which is initialized with $V(prop)$; the ordering of $Q$ is specified by parameter $O_q$

*DPVS* selects a variable in $Q$ and tries to find a counterexample with its first definition; if it fails it iteratively tries with the other definitions of the selected variable.

*DPVS* sets the color of conditional node $u$ to red (resp. blue) when condition of $u$ is set to true (false) in the current path. In other words, when the color is set to red (resp. blue) the right (resp. left) successor link of $u$ is cut off. $color[u]$ is initialized to blank for all nodes.

*DPVS* returns *Sol* which is either an instantiation of the input variables of $P$ satisfying constraint system $C$ or $\emptyset$ if $C$ does not have any solution. Solutions are computed by function *solve*, the finite domain solver of `Comet`[7]. Function *solve* is a complete decision procedure over the finite domains. Function $isfeasible$, used in line 27, performs only a partial consistency test. In other words, it detects some inconsistencies but not all of them. However, function $isfeasible$ is much more faster than function *solve*; this is the reason why we chose to only perform this test each time the constraints derived from the definition of a variable are added to the constraint store. This partial consistency check can either be done with the finite domain solver of `Comet` or with a the linear programming solver include in the `Comet` framework. However, to use the LP solver a linear relaxation of the constraint system has to be computed.

It is easy to show that *Sol*, the solution computed by *DPVS* is actually a counterexample. Indeed, these values of the input data satisfy the constraints generated from:

- *pred*, the required precondition;
- $\neg prop$ , the negation of a conjunct of the postcondition;
- one definition of all variables in $V(prop)$ and one definition of all variables (except the input variables) introduced by these definitions;
- all conditions required to reach the above mentioned definitions.

Thus, there exists at least one executable path whitch takes as input values *sol* and computes an output that violates the property *prop*.

Otherwise, when no solution can be found, we can state that there does not exist any input values that violate property *prop*; in other words that no counterexample can be found.

## 3. BENCHMARKS AND APPLICATIONS

In this section we describe the application examples. We start with some academic examples, then a well-known Traffic Alert and Collision Avoidance System (TCAS), and last

[7] `Comet`, a trade mark of Dynadec (see `http://dynadec.com/`) is an hybrid optimization platform.

**Algorithm 1** : *DPVS*

**Function** $DPVS(M, S, Q, O_q)$ returns *counterexample*

```
 1: if Q = ∅ then
 2:   return solve(S)
 3: else
 4:   x ← POP(Q, O_q)
 5:   for all u ∈ du[x] do
 6:     Cut ← FALSE; SAVE vector Color and Q
 7:     S_1 ← S ∧ const(def[x, u])
            % def[x, u] denotes the definition of x in block u
 8:     V_new ← V(def[x, u]) \ M
 9:     PUSH(Q, V_new, O_q); add(V_new, M)
10:     for all v ∈ anc_c[u] do
11:       if color[v] = blank then {%no branch is cut off}
12:         V_new ← V(condition[v]) \ M
13:         PUSH(Q, V_new, O_q); add(V_new, M)
14:         if dr[u, v] then {% Condition must be true}
15:           S_1 ← S_1 ∧ cons(condition[v]))
16:           color[v] ← red % Cut the right branch
17:         else {% Condition must be false}
18:           S_1 ← S_1 ∧ ¬ cons(condition[v])
19:           color[v] ← blue % Cut the left branch
20:         end if
21:       else
22:         if (color[v] = red ∧ dr[u, v])
                    ∨ (color[v] = blue ∧ ¬(dr[u, v])) then
            {%no branch is reachable}
23:           Cut ← TRUE
24:         end if
25:       end if
26:     end for
27:     if ¬Cut ∧ isfeasible(S_1) then
28:       result ← DPVS(M, S1, Q, O_q)
29:       if result ≠ ∅ then
30:         return result
31:       end if
32:     end if
33:     RESTORE vector Color and Q
34:   end for
35:   return ∅
36: end if
```

a real time industrial application component: a controler that drives several functions related to the flashing lights of a car.

## 3.1 Academic examples

### 3.1.1 Tritype program

The tritype program is a standard benchmark in test case generation and program verification since it contains numerous non-feasible paths: only 10 paths correspond to actual inputs because of complex conditional statements in the program. This program takes three positive integers as inputs (the triangle sides) and returns 2 if the inputs correspond to an isosceles triangle, 3 if they correspond to an equilateral triangle, 1 if they correspond to some other triangle, and 4 otherwise. The tritype program and its specification in JML[8] is shown in Figure 11 in the appendix. Note that in JML corresponds to the value returned by the program. Note also the role of local variable "trityp": It determines how many sides are equal and which are the equal sides.

We also consider two variations of the *Tritype* program:

1. *Triperimeter*: returns the perimeter of the triangle when the inputs correspond to a triangle, and $-1$ otherwise.
2. *Tritimes*: returns the product of the inputs.

Both *Triperimeter* and *Tritimes* have the same control flow graph as *Tritype*, and their specifications also have the same conditional form. The difference is the final value assigned to variable *trityp*. For example, in line 25 of Figure 11, *trityp* takes value $2 * j + i$ for *Triperimeter* and value $j * j * i$ for *Tritimes* because this line corresponds to the case where sides $j$ and $k$ are equal.

These two variations are more challenging than the *Tritype* program itself: *Triperimeter* returns a linear expression on the inputs and *Tritimes* returns a **non** linear expression on the inputs. On the contrary, *Tritype* returns a constant.

For these three programs, we also considered some faulty versions where an error has been introduced in the control structure (see line 22 in Figure 11).

### 3.1.2 Binary search

The second academic example illustrates difficulties of verifying and testing programs with arrays and loops. It is a binary search program that determines whether a value $v$ is present in a sorted array $t$. This program is depicted in Figure 10 in the appendix.

## 3.2 Traffic Alert and Collision Avoidance System

This application concerns a public software component of a Traffic Alert and Collision Avoidance System (TCAS). TCAS is an on-board aircraft conflict detection and resolution embedded system intended to alert the pilot to the presence of nearby aircraft. This critical system is well documented and well-known in the literature [18, 4]. We consider here the program and the specifications that were written by Arnaud Gotlieb [13, 14] starting from a preliminary version given in [19]. We summarize here their main characteristics.

The program contains 173 lines of $C$ code including nested conditionals, logical operators, type definitions, macros and

[8]See `http://www.cs.ucf.edu/leavens/JML/`

function calls. The main function under verification takes 14 global variables as inputs. 10 safety properties have been identified and formalized in the litterature (see [13] for a complete presentation of these properties). For instance, Figure 12 in appendix depicts the $C$ program with a specification to ensure that the "Safe advisory selection" property is satisfied. This property means that a downward resolution advisory is never issued when a downward manoeuvre does not produce an adequate separation from the nearby aircraft.

## 3.3 The Flasher Manager

This last benchmark is taken from a real time industrial software component[9]. It illustrates how we handle properties over several time periods. The complexity of the $C$ code is comparable to the complexity of TCAS benchmark; however, we have to check a property for several executions of the code.

### 3.3.1 Description of the module

The *Flasher Manager* is the controller that drives several functions related to the flashing lights of a car. The flashing lights serve several purposes:

1. Under normal operation, when the driver wishes to indicate a direction change, the `CBWS_HAZARD_R` or `CBWS_HAZARD_L` boolean inputs rise from 0 to 1. The corresponding light (driven by the `CMD_FLASHER_R` or `CMD_FLASHER_L` output respectively) shall then oscillate between an on/off state over a period of 3 time-units (typically 3 seconds). Then, when the input falls back to 0, the corresponding output light shall stop flashing. This is called the `Flashers_left` and `Flashers_right` functions.

2. The driver has the ability to lock and unlock the car from the distance using a RF-key. The state of the open and close buttons of the key is reported to Boolean inputs: `RF_KEY_UNLOCK` and `RF_KEY_LOCK` respectively. The manager has to indicate the state of the doors to the user using the following convention:

   - If the unlock key is pressed while the car is unlocked, nothing shall happen.
   - If the unlock key is pressed when the car is locked, both lights shall flash with a period of 10 time-units during 20 time-units (slow flashes). This is the `Warning_slow` function.
   - If the lock button is pressed while the car is unlocked, both lights shall go on for 10 time-units, and then shall go off.
   - If the lock button is pressed while the car is locked, both lights shall flash during 60 time-units with a period of 1 time-unit (quick flashes for a long time). This is the `Warning_fast` function. It is typically used to locate the car in an over-filled place.

3. Finally the driver has the ability to press the warning button. When a `WARNING` is present (reflected in the value of the `WARNING` input), both lights shall flash with a period of 3 time-units. This is called the `Warning` function.

Figure 6 shows a simplified Simulink model of the *Flasher Manager* (i.e. input/outputs) and Figure 7 shows a more detailed model.

[9]This example comes from a car manufacturer and has been provide by Geensys (see `http://www.geensys.com/?Home&l=en`.

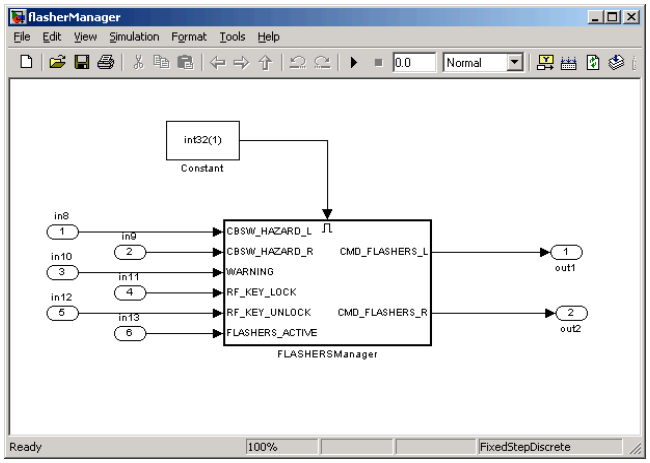


**Figure 6: Simplified Simulink model of the *Flasher Manager***

Some functions of the *Flasher Manager* are more important than others and must be served first. This results in some priority relations. We have been asked to check the following property: ***The lights should never remain lit*** ($p_1$)

### 3.3.2 Program under test

In order to check property ($p_1$), the Simulink model of the *Flasher Manager* is first translated into a $C$ function[10], named function $f_1$. Function $f_1$ involves 81 Boolean variables including 6 inputs and 2 outputs and 28 integer variables. Function $f_1$ contains 300 lines of code and mainly consists of nested conditionals including linear operations and constant assignments, as illustrated by the piece of code displayed in Figure 8.

```
and1_a=((Switch5==TRUE)&&(TRUE!=Unit_Delay3_a_DSTATE));
if ((TRUE==((and1_a-Unit_Delay_c_DSTATE)!= 0))) {
rtb_Switch_b=0;
}
else {
add_a = (1+Unit_Delay1_b_DSTATE);
rtb_Switch_b = add_a;
}
superior_a = (rtb_Switch_b>=3);
```

**Figure 8: Piece of code of the $f_1$ function**

Our aim is to check if there exists an input data sequence that violates the property ($p_1$) of the *Flasher Manager*. This property concerns the behaviour of the *Flasher Manager* for an infinite time period. Practically, we can only check a bounded version of property ($p_1$): we consider that the property is violated when the lights remain on for $N$ consecutive time periods. We thus introduce a loop (bounded by value $N$) that counts the number of times where the output of the *Flasher Manager* has consecutively been true. After the loop, if this counter is equals to $N$, then the property is violated in the sense that the output has remained true for $N$ consecutive time periods. The value of the bound $N$ is fixed as great as possible as shown in section 4; its maximal value is mainly determined by the capabilities of the tools.

[10]This translation is done with a proprietary tool of Geensys.

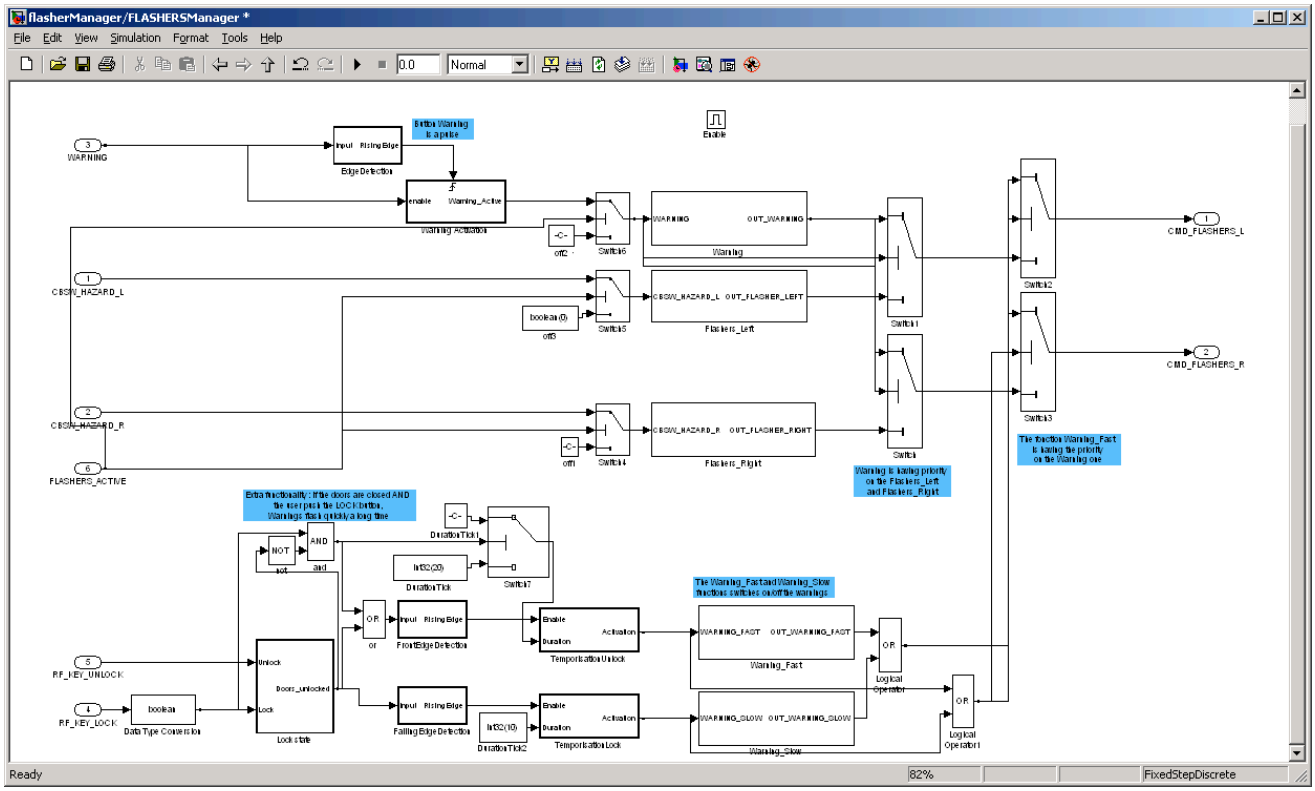


**Figure 7: Detailed Simulink model of the *Flasher Manager***

The part of the $C$ program that corresponds to this bounded version of property ($p_1$)is displayed in Figure 9.

```
// number of time where the output has been consecutively
//                                    true int count = 0;
// consider N periods of time
for(int i=0;i<N;i++) {
// call to f1 function to compute the outputs
// according to non deterministic input values
f1();
if (Model_Outputs4)
   // the output has been consecutively true one more time
   count++;
else
   // the output is not consecutively true
   count=0;
}
// if count is less than N, then the property is verified
assert (count<N)};
```

**Figure 9: C program under test**

# 4. EXPERIMENTS AND DISCUSSION

In this section, we report the experiments we have done to validate our approach and we discuss further works.

## 4.1 Tools

We compared performances of *DPVS* with CBMC, Z3, Euclide and CPBPV*.

As said before, CBMC[11] is one of the best bounded model-checkers. We used version 3.3.2 that calls the SAT solver *MiniSat2*.

Z3[12] is a state of art Satisfiability Modulo Theories (SMT) solver[20] developed at Microsoft Research. Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers. Z3 is integrated with a number of program analysis, testing, and verification tools from Microsoft Research. Experiments were performed with Z3 2.4.

CPBPV* is an optimized version of CPBPV [6, 7] which is implemented in *Comet*[13]. Like CPBPV it uses constraint stores to represent both the specification and the program, and to explore execution paths of bounded length over these constraint stores. However, contrary to CPBPV, it works on a compacted CFG. A preliminary bound propagation step is also performed.

Euclide [13] is constraint-based testing framework designed for structural test data generation, counterexample generation and partial program proving for critical C programs. We could not evaluate Euclide on the flasher manager application. Indeed, due to a bug, Euclide could not handle this C program[14]. Performances of Euclide on the TCAS

[11] see `http://www.cprover.org/cbmc`

[12] see `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

[13] Comet is a software platform for solving complex combinatorial optimization problems. Comet combines the methodologies used for constraint programming, linear and integer programming, constraint-based local search, and dynamic stochastic combinatorial optimization with a new, rich language for modeling and searching (see `http://dynadec.com/technology/`

[14] We have contacted the authors but they could not fix this

application are the one reported by the authors in [13]. The author's experiments were performed on a an Intel Core Duo 2.4GHz clocked PC with 2GB of RAM, a very similar computer to the one we used.

*DPVS* is implemented in `Comet`. We report here experiments for two variable orderings : LIFO and Heap. We tried different combinations of finite domain constraint solvers and linear programming solvers. We report results for two combinations:

- CP-CP: the finite domain constraint solver is used both to check the (partial) consistency at each node and to search a solution;
- LP-CP: A linear programming solver is used to check the (partial) consistency of a linear relaxation of the constraint system at each node, and the finite domain constraint solver is used to search a solution.

Experiments with *CBMC*, *Z3*, *CPBV∗* on all benchmarks but the flasher manager application were performed on an Intel Duo Core T8300 2.4GHz clocked PC with 2GB of RAM. Experiments on the *Flasher Manager* were performed on an Quadcore Intel Xeon X5460 3.16GHz clocked with 16Gb memory. All times are given in seconds.

OoM stands for "out of memory" whereas TO stands for "time out" in the different tables. The time out was set to 3 minutes for all benchmarks.

## 4.2 Experiments on academic examples

We start with the academic examples. Table 1 provides the experimental results for the *Tritype* program and its variations. These benchmarks are quite easy and all solvers found a counterexample for the faulty programs in almost no time. For the correct programs, proving that no counterexamples exist is more difficult. Indeed, all paths must be explored, contrary to the faulty versions where the resolution stops as soon as the first error is found. However, solvers that integrate a linear solver can do it for Tritype and Triperimeter.

The correct version of the non-linear *Tritimes* program is the most difficult to handle. Only the version of *DPVS* and *CPBPV∗* which combine LP and CP were able to verify the correct version of *Tritimes*. This can be explained by the following reasons.

1. Decisions are efficiently tested with a linear solver;
2. Decisions are incrementally added into the constraint store;
3. The *CP* solver which is called at the end of the paths takes benefit of the decisions that have been added to the contraint store, thanks to a substitution mechanism of sub-expressions.

Points (1) and (2) ensure that infeasible paths are quickly pruned. Let us explain point (3) on a small example. Consider the specification of an isosceles triangle, that is to say, $\backslash result = i * j * k$. Suppose decision $i = j$ has been taken on a path, and thus the value returned by the program is $i * i * k$. The constraint system will contain the three constraints $r = i * j * k$, $i = j$, $\neg(r = i * i * k)$[15], and the *CP* solver can easily detect the inconsistency.

bug yet.

[15]We do not detail here the SSA renamings.

Table 2 reports the results of the experiments on a correct version of the *Binary Search* program.

CBMC, Z3 and *DPVS* cannot handle this benchmark. CBMC and Z3 waste a lot of time in the unfolding process. The strategies used by *DPVS* are not well adapted for this very specific program. The LP-CP version of CPBPV* outperforms the other checkers. CPBPV* has a top-down approach and incrementally adds the decisions taken along a path. This strategy is particularly well adapted for the *Binary Search* program which has a strong precondition. This precondition combined with the decisions taken along some path have a strong impact on feasibility of the next conditions, and help to prune infeasible paths.

**Table 1: Tritype, Tritimes, Triperimeter (integer of 16 bits)**

| Program | CBMC 3.3.2 | Z3 2.3 | DPVS LIFO CP-CP | CPBPV* CP-CP | DPVS LIFO LP-CP | CPBPV* LP-CP |
|---|---|---|---|---|---|---|
| Trityp-OK | 0.161 | 0.26 | TO | TO | 0.781 | 0.304 |
| Trityp-KO | 0.012 | 0.01 | 0.087 | 0.127 | 0.030 | 0.009 |
| Triper-OK | 0.717 | 1.54 | TO | TO | 0.054 | 0.054 |
| Triper-KO | 0.015 | 0.02 | 0.002 | 0.027 | 0.014 | 0.018 |
| Tritm-OK | TO | TO | TO | TO | 1.048 | 0.865 |
| Tritm-KO | 0.050 | 0.15 | 0.070 | 0.002 | 0.029 | 0.009 |

**Table 2: Binary search (integers of 16 bits)**

| Length | CBMC 3.3.2 (MiniSAT2) | Z3 2.3 | DPVS LIFO LP-CP | CPBPV* LP-CP |
|---|---|---|---|---|
| 4 | 5.732 | 0.78 | 0.529 | 0.107 |
| 8 | 110.081 | TO | 35.074 | 0.298 |
| 16 | TO | TO | TO | 1.149 |
| 32 | TO | TO | TO | 5.357 |
| 64 | TO | TO | TO | 27.714 |
| 128 | TO | TO | TO | 153.646 |

## 4.3 Experiments on TCAS

Results for TCAS application are reported in Table 3. CBMC and Z3 have good results on this benchmark but the combination CP-CP of *DPVS* yields the best results. This can be easily explained by the fact that this benchmark is mainly a SAT problem (the program contains many Booleans, and integers can only take few values). That's also why the combination CP-CP of CPBPV* behaves better than the combination LP-CP of CPBPV* on this problem.

Other strategies of *DPVS* behave well. We did not report the results of the strategies which use a heap because the results are very similar to the one that use a LIFO queue.

Experiments on *TCAS* were performed for integers of 16 bits because we wanted to compare our results to those published in [14]. Euclide is much more slower than the other solvers but this may (partially) be due to the fact that the solver is implemented in a Prolog framework.

## 4.4 Experiments on the Flasher Manager

CBMC, Z3 and *DPVS* found counterexamples that violate the bounded version of property ($p_1$)They also generated data input sequences such that the flasher lights remain lit for $N$ consecutive periods of time.

**Table 3: TCAS (integers of 16 bits)**

| Property | CBMC 3.3.2 | Z3 2.3 | Euclide (SICtus Prolog) | DPVS LIFO CP-CP | CPBPV* CP-CP | DPVS LIFO LP-CP | CPBPV* LP-CP |
|---|---|---|---|---|---|---|---|
| P1A-OK | 0.101 | 0.06 | 0.7 | 0.021 | 0.173 | 0.028 | 8.960 |
| P1B-OK | 0.063 | 0.05 | 0.7 | 0.019 | 0.211 | 0.030 | 13.737 |
| P2A-OK | 0.097 | 0.05 | 0.6 | 0.020 | 0.175 | 0.029 | 9.168 |
| P2B-KO | 0.064 | 0.05 | 0.7 | 0.011 | 0.014 | 0.032 | 0.165 |
| P3A-KO | 0.061 | 0.05 | 5.4 | 0.008 | 0.036 | 0.044 | 0.615 |
| P3B-OK | 0.066 | 0.05 | 1.2 | 0.011 | 0.100 | 0.027 | 0.4431 |
| P4A-KO | 0.075 | 0.05 | 6.8 | 0.008 | 0.045 | 0.042 | 1.446 |
| P4B-KO | 0.060 | 0.05 | 2.7 | 0.008 | 0.014 | 0.036 | 0.254 |
| P5A-OK | 0.068 | 0.06 | 0.6 | 0.044 | 0.197 | 0.035 | 20.627 |
| P5B-KO | 0.065 | 0.05 | 1.0 | 0.015 | 0.014 | 0.029 | 0.195 |

**Table 4: Flasher Manager**

| N | CBMC 3.3.2 | Z3 2.3 | DPVS HEAP CP-CP | DPVS LIFO CP-CP | DPVS HEAP LP-CP | DPVS LIFO LP-CP |
|---|---|---|---|---|---|---|
| 5 | 0.134 | 0.15 | 0.026 | 0.032 | 0.565 | 0.953 |
| 10 | 0.447 | 0.5 | 0.055 | 0.068 | 2.484 | 4.134 |
| 20 | 1.766 | 3.29 | 0.118 | 0.149 | 9.081 | 15.320 |
| 30 | 4.231 | 7.83 | 0.194 | 0.248 | 20.847 | 35.066 |
| 50 | 12.92 | 27.89 | 0.345 | 0.458 | 57.797 | 96.693 |
| 75 | 32.747 | 61.83 | 0.602 | 0.842 | 137.104 | TO |
| 100 | 58.279 | 128.74 | 2.750 | 3.394 | TO | TO |
| 150 | 138.192 | TO | 1.552 | 2.365 | TO | TO |
| 200 | OoM | OoM | 6.003 | 8.082 | TO | TO |

For instance, here is a data input sequence that violates this property for 5 time periods:

```
[(0,1,0,0,0,1),(0,1,0,0,0,1),(0,1,0,0,0,1),
(0,0,1,0,0,1),(0,0,0,0,0,1)]
```

where $(0,1,0,0,0,1)$ is the value of inputs $in_1$ to $in_6$ for the first time period, $(0,1,0,0,0,1)$ the value of the inputs for the second time period and so on.

Table 4 shows that the CP-CP of *DPVS* outperforms the other approaches on the *Flasher Manager* application. The CP-CP Heap version of *DPVS* is able to generate counterexamples for instances up to 400 time periods before running out of memory.

CPBPV* did not manage to handle this application for instance with $n$ greater than 10.

Using a finite domain solver to check the partial consistency is much efficient on this application than using a linear programming solver on a linear relaxation of the constraints. Actually, the difference of performance comes not from the performances of the solvers but from the choice points introduced by the linear relaxation. Indeed, the linear relaxation required by the LP solver introduces many choice points. Let us explain this point on a small example. Consider a test such that $x == y$, the negation of this test corresponds to the constraint $x! = y$ which introduces two choice points: $x < y$ and $x > y$.

Furthermore, the domains of the integer variables are small for this application, and the propagation step we perform reduces the bounds of the domain. Thus, consistency checks with CP are very efficient.

## 4.5 Discussion and further works

First experiments with *DPVS* are very encouraging.

*DPVS* behaves well on academic examples and obtains better results than Z3 and CBMC on two real applications. Generating test cases for realistic time periods is a critical issue in real time applications. For the *Flasher Manager* application, *DPVS* generated counterexamples for much significant time period than Z3 and CBMC.

These results are impressive since *DPVS* is still a (slow) academic prototype whereas Z3 and CBMC are state of art solvers. *DPVS* also outperforms Euclide on the TCAS application. Of course, other experiments on real applications are required to refine and validate the proposed approach.

Further work concerns also the extension of our prototype. There are many restrictions on the C and Java program that the current prototype can handle. Especially, we only handle run-time error-free programs, that's to say we do not handle programs the execution of which yield errors like dividing by zero or exceptions. We handle arrays but input data are restricted to Booleans and integers. We are working on an interface between the COMET solver and our own floating point number solver[3]. Expressions can contain the following operators: $+, -, *, /$, &&, $||, =, ==, !$ (but neither $\hat{}$ , `xor` nor library functions like, `sin`, `cos`,...). However, we are working on a new version with these capabilities to be able to evaluate the proposed approach on a larger class of programs.

# 5. REFERENCES


[1] Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *LOPSTR 2008(Logic-Based Program Synthesis and Transformation), Revised Selected Papers*, volume 5438 of *LNCS*, pages 4–23. Springer, 2008.

[2] Thomas Bochot, Pierre Virelizier, Hélène Waeselynck, and Virginie Wiels. Model checking flight control systems: The Airbus experience. In *ICSE 2009 (31st International Conference on Software Engineering),Companion Volume*, pages 18–27. IEEE, 2009.

[3] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Softw. Test., Verif. Reliab.*, 16(2):97–121, 2006.

[4] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzè. Using symbolic execution for verifying safety-critical systems. In *ESEC / SIGSOFT FSE*, pages 142–151, 2001.

[5] Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2006.

[6] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV: A Constraint-Programming Framework for Bounded Program Verification. In *CP 2008 (14th International Conference on Principles and Practice of Constraint Programming)*, volume 5202 of *LNCS*, pages 327–341. Springer, 2008.

[7] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV: A Constraint-Programming Framework for Bounded Program Verification. *Constraint*, http://www.springerlink.com/content/j22226282p0v4220, 2010.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[9] Tristan Denmat, Arnaud Gotlieb, and Mireille Ducassé. Improving constraint-based testing with dynamic linear relaxations. In *Proc. of ISSRE, The 18th IEEE International Symposium on Software*, pages 181–190. IEEE Computer Society, 2006.

[10] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[11] Gordon Fraser, Franz Wotawa, and Paul Ammann. Testing with model checkers: a survey. *Softw. Test., Verif. Reliab.*, 19(3):215–261, 2009.

[12] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223. ACM Press, 2005.

[13] Arnaud Gotlieb. Euclide: A Constraint-Based Testing Framework for Critical C Programs. In *ICST 2009, Second International Conference on Software Testing Verification and Validation, 1-4 April 2009, Denver, Colorado, USA*, pages 151–160. IEEE Computer Society, 2009.

[14] Arnaud Gotlieb. TCAS software verification using constraint programming. *The Knowledge Engineering Review, (Accepted for publication)*, 2010.

[15] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA,International Symposium on Software Testing and Analysis*, pages 53–62, 1998.

[16] Paula Herber, Florian Friedemann, and Sabine Glesner. Combining model checking and testing in a continuous hw/sw co-verification process. In *TAP 2009(Third International Conference on Tests and Proofs)*, volume 5668 of *LNCS*, pages 121–136. Springer, 2009.

[17] Daniel Jackson and Mandana Vazir. Finding bugs with a constraint solver. In *ISSTA, International Symposium on Software Testing and Analysis*, pages 14–25. ACM Press, 2000.

[18] Carolos Livadas, John Lygeros, and Nancy A. Lynch. High-level modeling and analysis of tcas. In *IEEE Real-Time Systems Symposium*, pages 115–125, 1999.

[19] Carolos Livadas, John Lygeros, and Nancy A. Lynch. High-Level Modeling and Analysis of TCAS. In *IEEE Real-Time Systems Symposium*, pages 115–125, 1999.

[20] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Challenges in satisfiability modulo theories. In *RTA'07, 18th International Conference onTerm Rewriting and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2007.

[21] N.Williams, B.Marre, P.Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing -EDCC'05*, pages 281–292, 2005.

[22] Nguyen Tran Sy and Yves Deville. Automatic test data generation for programs with integer and float variables. In *ASE (16th IEEE International Conference on Automated Software Engineering)*, pages 13–21. IEEE Computer Society, 2001.

# APPENDIX

```
/*@ requires (\forall int i; i>=0
@           && i<t.length-1;t[i]<=t[i+1])
@ ensures
@  (\result!=-1 ==> t[\result] == v) &&
@  (\result==-1 ==> \forall int k;
@                     0<=k<t.length;t[k]!=v)
@*/
1 static int binary_search(int[] t, int v) {
2    int l = 0;
3    int u = t.length-1;
4    while (l <= u) {
5      int m = (l + u) / 2;
6      if (t[m]==v)
7          return m;
8      if (t[m] > v)
9         u = m - 1;
10      else
11        l = m + 1;
12   }
13   return -1;
14 }
```

**Figure 10: The Binary Search Program.**

```
/*@ requires (i>=0)&&(j>=0)&&(k>=0);
@ ensures
@   ((i+j<=k)||(j+k<=i)||(i+k<=j)) ==> \result == 4 &&
@   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&((i==j)&&(j==k))
                          ==> \result == 3 &&
@   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&!((i==j)&&(j==k))
@        &&((i==j)||(j==k)||(i==k)) ==> \result == 2 &&
@   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&!((i==j)&&(j==k))
@        &&!((i==j)||(j==k)||(i==k)) ==> \result == 1;
@*/
1  public static int tritype(int i, int j, int k){
2   int trityp ;
3   // not a triangle
4   if ((i==0)||(j==0)||(k==0)) trityp = 4 ;
5   else {
6     trityp = 0 ;
7     if (i==j) trityp = trityp + 1 ;
8     if (i==k) trityp = trityp + 2 ;
9     if (j==k) trityp = trityp + 3 ;
10    if (trityp==0){
11       // triangular inequality not verified
12       if ((i+j <= k)||(j+k <= i)||(i+k <= j)) trityp = 4 ;
13       else trityp = 1 ; // any triangle
14    }
15    else {
16      if (trityp > 3) trityp = 3 ; // equilateral
17      else
18        //i=j and triangular inequality verified
19        if ((trityp==1)&&(i+j>k)) trityp = 2 ;
20        else
21          //i=k and triangular inequality verified
22          if ((trityp==2)&&(i+k>j)) trityp = 2 ;
                          //ERROR if ((trityp==1)&&(i+k>j))
23          else
24            //j=k and triangular inequality verified
25            if ((trityp==3)&&(j+k>i)) trityp = 2 ;
26            else trityp = 4 ; // not a triangle
27          }
28      }
29    return trityp;
30  }
```

**Figure 11: Tritype program**

```
/@ ensures
 (((Up_Separation>=Positive_RA_Alt_Thresh_2)
 && (Down_Separation<Positive_RA_Alt_Thresh_2))==>(\result!=2));
@/

int alt_sep_test(int Cur_Vertical_Sep, int High_Confidence,
 int Two_of_Three_Reports_Valid, int Own_Tracked_Alt,
 int Own_Tracked_Alt_Rate, int Other_Tracked_Alt,
 int Positive_RA_Alt_Thresh_0, int Positive_RA_Alt_Thresh_1,
 int Positive_RA_Alt_Thresh_2, int Positive_RA_Alt_Thresh_3,
 int Up_Separation, int Down_Separation, int Other_Capability,
 int Climb_Inhibit) {
   int alt_sep = 0;
   int Non_Crossing_Biased_Climb;
   int Non_Crossing_Biased_Descend;
   int Inhibit_Biased_Climb;
   int ALIM;
   int Alt_Layer_Value = 2;
   int Other_RAC = 0;

   if(Climb_Inhibit != 0)
  Inhibit_Biased_Climb = Up_Separation + 100;
   else
  Inhibit_Biased_Climb = Up_Separation;

   if( Alt_Layer_Value==0 ) {
  ALIM = Positive_RA_Alt_Thresh_0; }
   if( Alt_Layer_Value==1 ) {
  ALIM = Positive_RA_Alt_Thresh_1; }
   if( Alt_Layer_Value==2 ) {
       ALIM = Positive_RA_Alt_Thresh_2; }
   if( Alt_Layer_Value==3 ) {
   ALIM = Positive_RA_Alt_Thresh_3; }

   if (Inhibit_Biased_Climb>Down_Separation){
      if(!(Own_Tracked_Alt<Other_Tracked_Alt) ||
        ((Own_Tracked_Alt<Other_Tracked_Alt) &&
                                 (!(Down_Separation>=ALIM))))
    Non_Crossing_Biased_Climb = 1;
      else
    Non_Crossing_Biased_Climb = 0;
   }
   else {
      if((Other_Tracked_Alt<Own_Tracked_Alt)
         && (Cur_Vertical_Sep>=300 ) && (Up_Separation>=ALIM))
     Non_Crossing_Biased_Climb = 1;
      else
     Non_Crossing_Biased_Climb = 0;
    }

    if (Inhibit_Biased_Climb>Down_Separation){
       if((Own_Tracked_Alt<Other_Tracked_Alt) &&
                                      (Cur_Vertical_Sep>=300 )
          && (Down_Separation>=ALIM))
          Non_Crossing_Biased_Descend = 1;
       else
     Non_Crossing_Biased_Descend = 0;
       }
    else {
       if(!(Other_Tracked_Alt<Own_Tracked_Alt)
         || ((Other_Tracked_Alt<Own_Tracked_Alt)&&
                                       (Up_Separation>=ALIM)))
     Non_Crossing_Biased_Descend = 1;
       else
     Non_Crossing_Biased_Descend = 0;
    }

    if ((High_Confidence==1)&&(Own_Tracked_Alt_Rate<=600 )&&
     (Cur_Vertical_Sep>600 )&&(((Other_Capability==1)&&
     (Two_of_Three_Reports_Valid==1)&&(Other_RAC==0)) ||
      !(Other_Capability==1))) {
    if((Non_Crossing_Biased_Climb==1)
        &&(Own_Tracked_Alt<Other_Tracked_Alt)&&
                            (Non_Crossing_Biased_Descend==1))
           alt_sep = 0 ;
    else
          if ((Non_Crossing_Biased_Climb==1)&&
                         (Own_Tracked_Alt<Other_Tracked_Alt))
            alt_sep = 1 ;
          else if ((Non_Crossing_Biased_Descend==1)&&
                         (Other_Tracked_Alt<Own_Tracked_Alt))
            alt_sep = 2 ;
          else
              alt_sep = 0 ;
 }
 return alt_sep;
}
```

**Figure 12: TCAS program with specification of property P1A**